LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-609323

Grafting AMR Capability Into the Diffusion Package of an Existing ALE Code

L. Howell

January 3, 2013

Nuclear Explosive Code Development Conference
Livermore, CA, United States
October 22, 2012 through October 26, 2012

Disclaimer

(U) Grafting AMR Capability Into the Diffusion Package of an Existing ALE Code

[1]**Louis Howell**
[1]*Lawrence Livermore National Laboratory, Livermore, CA*

Abstract

This paper describes the addition of parallel adaptive mesh refinement (AMR) capability to diffusion physics packages in an existing multi-block ALE code. As this code is in production use and under active development, it was necessary to develop a compatible AMR implementation without invalidating, changing, or adding overhead to the non-AMR portions of the algorithm. Issues discussed here include AMR discretization and data structures, multiblock complications, linear solver support, parallel scaling, and performance. Particular care was required to find a robust AMR discretization for the flux limiter. (U)

Introduction

Adaptive mesh refinement requires significant software infrastructure support and influences algorithmic decisions in major ways. It is typical, then, for an AMR code to be designed with AMR in mind from the beginning. The prospect of adding AMR to a non-AMR multiphysics simulation code suggests a thorough re-write, in which a few code fragments might survive but the basic code design would have to be changed from the ground up. It would not an incremental step in the code's evolution.

But that is exactly what several collaborators and I have been doing in our current project. We are fortunate in that the base code we are working from already resembles a structured AMR code in a number of respects. It is because of these similarities that the task is even possible, let alone practical. In a structured AMR code (cf. [2]) the mesh is decomposed into regular patches—rectangles in 2D—at varying resolutions. The non-AMR base code we are revising is also built on a mesh decomposed into rectangular patches, at large scale because it is a multiblock code and at small scale for distribution over a parallel machine. These patches communicate through filling ghost cells, and the communication patterns and infrastructure closely resemble what would be needed within a single level of refinement in an AMR mesh. We use the SAMRAI [10] AMR infrastructure to support the more complicated communication patterns needed between different levels of refinement. The initial stages of the extension to AMR, with particular focus on ALE hydrodynamics, are documented in [7]. This paper extends that work to diffusion physics and the coupling to the *hypre* [9] parallel linear solver package.

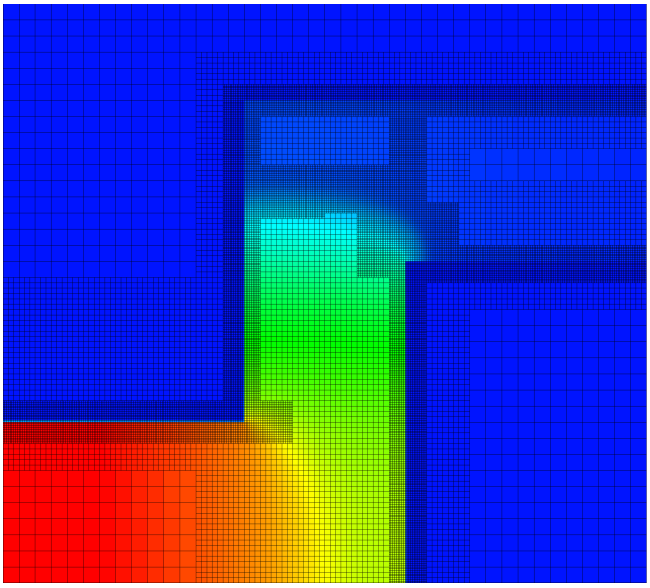

Figure 1: Detail from the crooked pipe test problem showing radiation temperature.

As an example, Fig. 1 shows a portion of the crooked pipe test problem ([4] and [3]) with refined patches in regions where the radiation field is changing most rapidly. Three levels of refinement are shown including the coarse base level. The patches making up each level are not directly visible; however, one criterion for all aspects of the algorithm development is that the computed solution should not depend on the way a refined portion of the mesh is broken into patches, so in that sense the exact decomposition does not matter. The refinement criterion used here is based on a scaled second derivative of radiation temperature. (Refinement criteria are under user control and can be constructed based on many different characteristics of the simulation.)

In Fig. 2 we see diffusion across a multiblock mesh with an arbitrary collection of refined patches. It is worth noting here that the mesh is not orthogonal, it includes reduced connectivity points, and refined patches may be adjacent to those points. The computed solution is smooth and does not show artifacts at either AMR or block boundaries. The refined patches were added for illustration purposes and were not based on features of either the problem or the solution. 3D multiblock grids are also supported, and such grids may include reduced connectivity edges. Enhanced connectivity points and edges are also supported with the minor constraint that the interfaces between different refinement levels cannot touch these points. We expect to lift this constraint in the near future.

Close examination of both of these figures will show that refinement in each direction is by a factor of 3. The current algorithm for ALE hydrodynamics requires odd refinement ratios and so 3 is by far the most common. The code supports other odd ratios, though, and in principle does not require the ratio to be the same in all directions. The infrastructure currently supports anisotropic refinement only for single-block grids, though multiblock support may be added in the future. The diffusion package described in this paper

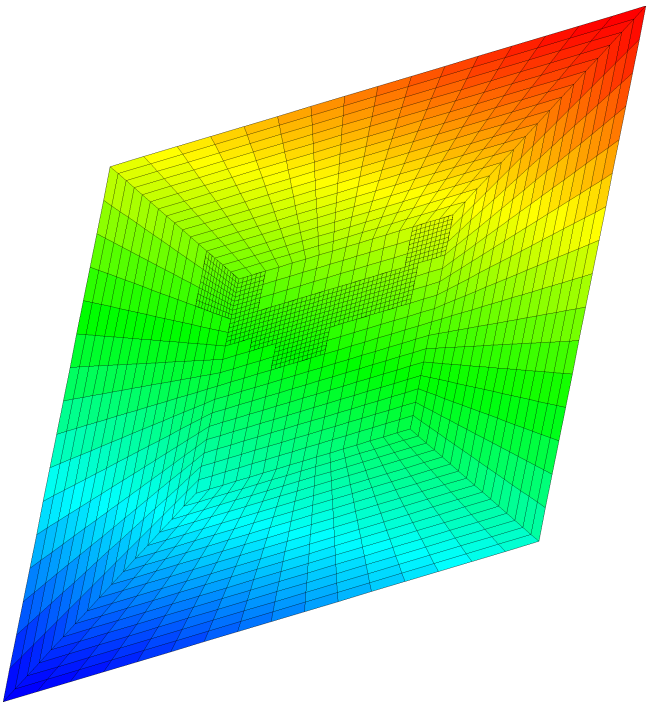

Figure 2: Diffusion across a non-orthogonal multiblock AMR mesh.

is being upgraded to support anisotropic refinement as this is being written.

It is important to bear in mind the constraints of the AMR conversion process for this code. The code is in production use, has been extensively validated, and is still actively under development. So in adding AMR support we are not to change the discretizations used in uniform parts of the mesh, we should not change the results, and we may change the data structures and implementation details only to the minimum necessary to insert AMR functionality. Finally, AMR adds significant overhead to run times, and as there are problems that will not benefit from AMR we must maintain the ability to run the code without AMR turned on and without adding significant overhead to such problems.

Diffusion in the Base Code

For diffusion we are again fortunate in that the base code uses a diffusion discretization [6] that extends to AMR in a reasonably natural way. This scheme, which I will refer to as 'Pert', is used in the code for several different diffusion processes including radiation diffusion and heat conduction. It was designed to be robust on very distorted meshes. The merits or justification of the scheme itself are not our concern here, only the issue of extending it to AMR.

Pert is a finite volume scheme, which makes it straightforward to match up fluxes at each coarse-fine

interface to ensure conservation. It cleanly separates out the roles of diffusion coefficients (that depend on the physics), metrics (that depend only on the mesh geometry), and finite differences (that are defined in computational index space). Only the finite differences need be explicitly reconstructed to be AMR-aware. (For some physics packages there is also a flux limiter, which has a stencil of its own and modifies the diffusion coefficient. I'll discuss the limiter later.)

Here is a very brief description of the Pert discretization: Let K, L, be the coordinates in index space in 2D, and consider the diffusion flux through the $+K$ side of a 2D cell centered at (K, L). In Cartesian coordinates this flux is

$$\mathbf{F} \cdot \mathbf{A}_K^+ = -\left(D/J\right)_{K+\frac{1}{2},L} \left[\left(\mathbf{R}_L \cdot \mathbf{R}_L\right) \phi_K - \left(\mathbf{R}_K \cdot \mathbf{R}_L\right) \phi_L\right]_{K+\frac{1}{2},L}, \tag{1}$$

where ϕ is the quantity being diffused, D is the physical diffusion coefficient, and $\mathbf{R}_K$, $\mathbf{R}_K$, and J are metric quantities. The metrics are defined in terms of finite differences of the physical coordinates x and y with respect to K and L. The main factors to focus on here are ϕ_K and ϕ_L, which are differences in index (K, L) space. The base code discretizes Eq. 1 as three terms:

$$\begin{aligned} \mathbf{F} \cdot \mathbf{A}_K^+ \quad = \quad & C_{K^+,L} \left[\phi_{K+1,L} - \phi_{K,L}\right] + \\ & \frac{1}{2} C_{K^+,L^+} \left[\phi_{K+1,L+1} + \phi_{K,L+1} - \phi_{K+1,L} - \phi_{K,L}\right] - \\ & \frac{1}{2} C_{K^+,L^-} \left[\phi_{K+1,L} + \phi_{K,L} - \phi_{K+1,L-1} - \phi_{K,L-1}\right], \end{aligned} \tag{2}$$

where the first line approximates the ϕ_K term as a normal difference across the $+K$ face, $C_{K^+,L}$ combines the relevant metrics and the diffusion coefficient for this face, and the second and third lines similarly approximate the ϕ_L term in terms of transverse differences centered at the nodes at either end of the face.

The final step in constructing the Pert stencil is to combine Eq. 2 with the fluxes for the other three faces of the 2D cell. Normal differences are unique to each face, so $C_{K^+,L}$ becomes the matrix coefficient connecting this cell (K, L) with cell $(K + 1, L)$. Transverse differences for adjacent faces combine in such a way that the contributions for orthogonal neighbors cancel out, leaving, for example, C_{K^+,L^-} as the matrix coefficient connecting to cell $(K + 1, L - 1)$.

The diffusion matrix row for a 2D cell thus reduces to 8 coefficients for the 8 surrounding cells, plus a coefficent on the matrix diagonal. The 3D stencil is more complicated, but can be decomposed into stencils similar to the 2D version in each of the three coordinate planes, yielding a 19-point discretization with no direct coupling along 3D diagonals. The base code computes and stores those 9 or 19 coefficients for each cell, saving memory by taking advantage of matrix symmetry.

AMR Diffusion Discretization

The key observation for the AMR implementation is that, even though the various C coefficients are stored in the code for use as matrix coefficients, they can *also* be interpreted as coefficients for elemetary finite difference expressions as shown in Eq. 2. These come in two types: normal differences across faces and "corner" differences (around nodes in 2D, edges in 3D). AMR stencils are then defined in three steps:

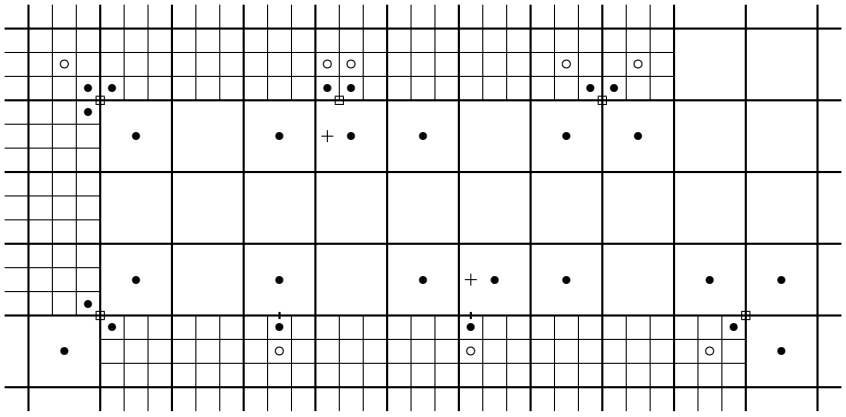

Figure 3: 2D stencils for AMR finite differences at various points along a coarse-fine interface, including both face- and node-centered cases. Open circles mark optional values that can be included to make some of the differences second-order. Crosses mark transverse interpolation points

1. Around the sides of each fine patch, physical coordinates and diffusion coefficient information are interpolated into ghost regions as appropriate. From these we construct "phony" matrix coefficients equivalent to those in the interior but corresponding to differences across the coarse-fine interface. These fit naturally into existing data structures because the base code already uses ghost regions to communicate with neighboring patches at the same refinement level. This step leverages existing capabilities in SAMRAI and the base code and requires little additional implementation.

2. AMR versions of the two kinds of elementary differences are constructed at faces and nodes at the coarse-fine interface. As these replace simple differences in computational space, they ignore the metrics for the physical mesh—those metrics are encapsulated in the phony matrix coefficients. This step is complicated and is explored in more detail below.

3. The flux across each coarse cell face at the coarse-fine interface is defined to be equal to the sum of the fluxes across the coinciding fine faces. (This makes the system conservative, but it does not make the matrix symmetric.) Coarse fluxes across transverse faces touching the coarse-fine interface are also modified, but only where they involve finite differences that cross the coarse-fine interface and therefore need to be replaced with AMR equivalents.

The AMR stencil design started with the normal (face-centered) differences, which could be based on prior implementations for 5-point diffusion stencils. For a detailed description of these differences see [1]; more recent implementations include [5] and [8]. In Fig. 3 these are represented by the two center cases on the bottom row. They are constructed in two steps: First the values in coarse cells are interpolated in the direction transverse to the interface, if necessary, to obtain a value aligned with the fine cell center (this is the point marked with a cross in the example at center-right). Using three coarse values as shown makes this step second-order accurate. Then the normal difference is constructed using this interpolant, the fine cell value, and an optional additional value from an interior fine cell to make the normal difference

second-order. Of the two examples shown in the figure the one on the left skips the transverse interpolation step because a coarse cell center is already aligned with the fine cell center.

It is important to keep in mind that when I say "values" are being interpolated, I'm really talking about unknowns in the linear system. We don't have numerical values to assign to those locations when we're building up the AMR stencils. Instead, what really happens is that the interpolation formulae are intermediate quantities incorporated into the discretizations defined at each cell along a coarse-fine interface. In the next section I will describe how those discretizations are assembled and passed into *hypre* to define the composite linear system.

None of the prior AMR diffusion implementations I had worked with included corner coupling, which for the Pert stencil adds transverse differences centered at nodes. At hanging nodes (fine interface nodes that are not nodes of the coarse mesh) I could use a construction analogous to that for the normal differences—an example appears in the center of the top row in Fig. 3. The center coarse cell is already aligned with the fine cells directly above it. Horizontal interpolation from coarse cells to the point marked with a cross gives us a value aligned with the other fine cells. Vertical interpolation in each column then gives a value on the coarse-fine interface (not shown), and a horizontal difference between these two interpolants is the desired transverse difference.

Interface nodes that are also nodes of the coarse mesh admit many more configurations based on the arrangement of coarse and fine quadrants around the node in question. The other four examples in Fig. 3—appropriately arranged around the corners of the figure—show the cases that can occur in 2D. Extending the approach from the previous paragraph to these corner cases proved very awkward. Instead, I adopted the stencil patterns shown in the figure, each of which is based on differences along two crossed diagonals. This required separate code for each corner case shown. Fortunately, in 3D the equivalent differences are built around edges, not around nodes, and so are based on the 2D cases shown in each of the three coordinate planes. It is not necessary to construct stencils around nodes in 3D, which would have involved a great many more special cases.

The end result is a rather ad hoc collection of stencils that take on a variety of different forms at different points of the coarse-fine interface. No numerical difficulties have been observed from this approach, except that the optional higher-order stencil versions (using the open circles in Fig. 3) proved vulnerable to oscillations and have not been used much in practice. Even though the method works, though, the implementation is ugly and there is at least one more practical drawback: the corner cases implemented with diagonal differences do not extend easily to handle anisotropic refinement. As this paper is being written I have been implementing an alternative, more systematic, approach to the AMR discretization based on a scheme similar to the one I'm using for the flux limiter, which is described below. This new implementation supports anisotropic refinement and may also improve performance.

hypre Semi-structured Interface

We use the *hypre* [9] parallel linear solver library for solving the diffusion equations implicitly. *hypre* is already used in the base code, but for AMR I had to write a new driver. In [5] and [8] *hypre* is used for solving on a single AMR level at a time, as part of a multi-stepping algorithm where finer levels advance

with smaller timesteps than coarser levels. The AMR extension of the current code, though, advances all levels at the same timestep, and so the solver has to couple all levels together in a single linear system.

The appropriate *hypre* interface for this sort of mesh is the *semi-structured* interface. The *structured* interface handles meshes consisting of rectangular patches in a single index space. The semi-structured interface extends this to multiple index spaces with arbitrary connectivity between them. It was designed for multiblock grids, but for AMR we go further and treat each (block,level) pair as a separate index space. Connections between blocks at the same level of refinement are handled directly by *hypre* primitives, but AMR connections are more complicated and must be fed into *hypre* one entry at a time.

For an example, see Fig. 2. There are five blocks and two levels of refinement, and so a potential total of ten "parts" or separate index spaces. The grid configuration shown in the figure has refined patches in only two of the blocks, though, so three of the ten parts are empty. The AMR stencils must support connections between levels that may also be connections to other blocks, with the associated index rotations and the additional complications due to reduced connectivity points.

AMR stencils come in many variations depending on where a fine cell is positioned within its containing coarse cell, whether it is at a corner of its fine patch, how other adjacent fine patches are arranged around that corner, and so on. It is useful to build up some of these stencils out of simpler components, such as interpolation from the coarse mesh. Neither the current *hypre* interface nor the SAMRAI library provide direct support for the necessary operations. Limitations of the *hypre* interface also require significant preprocessing to transform stencils from an AMR-based form into a form appropriate for the solver:

1. *hypre* distinguishes between stencil connections, which are the regular connections defined for any interior cell in a uniform section of the mesh (including the stencil-like connections to cells of the same level across block boundaries), and non-stencil connections, which are the additional entries that must be added for special purposes such as AMR. It is an error to add a special entry for a connection that is already in the stencil. So when processing the AMR discretizations the driver software must recognize which entries are to cells already in the interior stencil and which are not.

2. *hypre* views the special entries as additions to a connection graph. Each additional connection may only be added once. The AMR discretizations, though, are built up out of simpler primitive differences and so may revisit a given neighbor cell multiple times. The driver code must then remember which connections have been added so as not to add duplicates.

3. When modifying the coefficient associated with a connection (either stencil or special), *hypre* does not address the connection by its target cell location but rather by the order in which the connection was added to the graph. So we can't say "Add this amount to the entry connecting from cell (i, j) to cell (m, n)," we must say "Add this amount to the eleventh connection defined for cell (i, j)." As the pattern of connections is different for cells at different locations along the coarse-fine interface, the driver code must keep track of the the order in which connections were added for every interface cell.

4. *hypre* allows graph connections and their associated coefficients to be entered only from the processor that owns the cell in question. So even though parts of the coarse-cell AMR stencils are based on fluxes computed on neighboring fine cells, the coarse cells must recompute those fluxes in

order to enter them into *hypre* from the coarse-cell processor. Those fluxes depend on fine-cell diffusion coefficients, so those coefficients must be communicated to the processor owning the coarse cells in question.

To support these manipulations I have implemented a number of new data structures and operations using SAMRAI primitives. The *hypre* team is also working on AMR extensions, but it is not clear when these will be ready, how general they will be, or how many of the above list of difficulties they will address. So for the moment we will stick with the SAMRAI-based interface described here.

For each cell, coarse or fine, along a coarse-fine interface I define an object called an `AuxVar` that can hold a linked list of connections to other cells, including information about the block and the refinement level for each of those cells (but *not* an identifier for a specific grid patch, which is not needed by *hypre*). Each `AuxVar` can be thought of as preparing a single matrix row associated with a particular cell. These `AuxVars` are grouped into structures defined around the faces of each grid (for use on the fine side of the interface) and into other structures defined on the intersections of finer grid faces with the current grid (for use on the coarse side). The coarse side structures also have data structures for holding diffusion coefficients communicated from finer grids, and both fine and coarse structures have information to help determine the arrangement of coarse and fine regions surrounding each cell.

In addition to the `AuxVars` defining the matrix rows themselves, it is useful to have others that define intermediate quantities. These include the transverse interpolants discussed in the previous section (the "cross" points in Fig. 3) as well as both differences and interpolants normal to the interface. (The new stencil implementation I'm currently working on is based on interpolants located at fine ghost cells which can be managed by the same mechanism.) The way this works is that entries added to an `AuxVar` need not point directly to cell locations, they may instead point to other `AuxVars` as intermediate quantities.

Once the `AuxVar` for a matrix row has been fully constructed, but before it can be used, it must be collapsed and translated. Collapsing means that all intermediate quantities are replaced by their constituent entries and all entries with the same target cell are combined, leaving the connection list with a single entry per matrix column. Translation means performing the necessary conversions to transform entries connecting one block to another.

There are a number of subtleties to this process that I should at least mention even though I'm not going to describe them in tedious detail. Block translation is only appropriate for non-stencil entries, because *hypre* does its own internal translation for what it considers to be stencil entries. When a coarse-fine interface corresponds to a block boundary near a reduced-connectivity point, the coarse and the fine regions each see the block in the third "quadrant" as an orthogonal neighbor, not as a diagonal neighbor. That is, they disagree about where the third block fits into their local Cartesian index space. When stencils extend into this third block we have to be very careful to get the block translations right. With 3D stencils this gets even worse. These add additional interpolation in the third dimension (a detail that I skipped over in the previous section). The result is that some stencils can reach all the way around a reduced connectivity edge, landing back in the original block, where they have to be distinguished from stencil entries and other graph entries that did not wrap around the singularity.

So far I've been describing the coarse-fine stencils as input to *hypre*, but in some packages it is also useful to be able to evaluate them directly as a matrix-vector multiply. This changes the block translation requirements because of the mechanisms we rely on for parallel communication. For communication within a refinement level, or from coarse to fine, we have the native code or SAMRAI fill ghost cells around the faces of each patch. (For the coarse-to-fine case we use piecewise-constant interpolation so that the actual coarse values will be available.) For communication from fine to coarse we re-use the data structures prepared for communicating fine diffusion coefficients to a coarser grid, that were described above. Each local grid patch then holds in its own index space all of the data communicated from its neighbors. We might think then that there would be no need for further block translations as these had already been done. This is almost true, but not quite. The exceptions are those stencils that interact with reduced connectivity points and edges, which I alluded to above. To get an index location right for a coarse-side stencil (which starts out in the index space for the coarse block), it can be necessary to first translate it into the fine block, then into the third block if that is where it actually lives, then finally back into the coarse block. The cell location does not actually change. The coordinate system used to describe it changes three times, though, and in the end the index is apparently in a different location in the coordinate system it started in. The explanation is that the starting location is in a coordinate system extended in one direction around the singularity, while the final result is the same index in coordinates extended the opposite way around the singularity to match the data filled into ghost cells.

I have described a completed `AuxVar` as containing a list of connections to other cells, each with an associated coefficient. In the original implementation this was essentially true. But these objects are complicated and expensive to build. For setting up a call to a linear solver the overhead isn't too bad, since the solver itself is expensive to use. But for an explicit matrix-vector multiply the cost of building the interface stencils could be the dominant expense.

The solution was to re-use structures as much as possible. The connections themselves remain the same so long as the AMR mesh layout doesn't change, which means they can be re-used for several timesteps and for all of the physics packages that rely on the Pert discretization. (For multigroup diffusion that could mean a large number of energy groups.) The coefficients, though, change with every use. So the change to the code is that each `AuxVar` manages a list of connections, and each connection itself has a list of coefficient locations (and their weights) that it depends on. This adds a bit more complexity to the `AuxVars`, but tests show that refilling them with coefficients is roughly ten times faster than building them in the first place. Some performance results are included in the final section of this paper.

Flux limiter

Some physics packages include a limiter to keep fluxes within a valid range when diffusion coefficients become large. In this regime the diffusion approximation begins to fail. The flux limiter is an attempt to impose sanity on a solution that otherwise would become completely nonphysical. (In the case of radiation diffusion, the limiter prevents energy from moving faster than light.) There are various forms of flux limiter included in the base code but all of them reduce excessively large diffusion coefficients based on some function of a scaled gradient of the solution, $|\nabla\phi|/\phi$.

The straightforward way to evaluate this quantity with AMR is to use SAMRAI to interpolate the solution

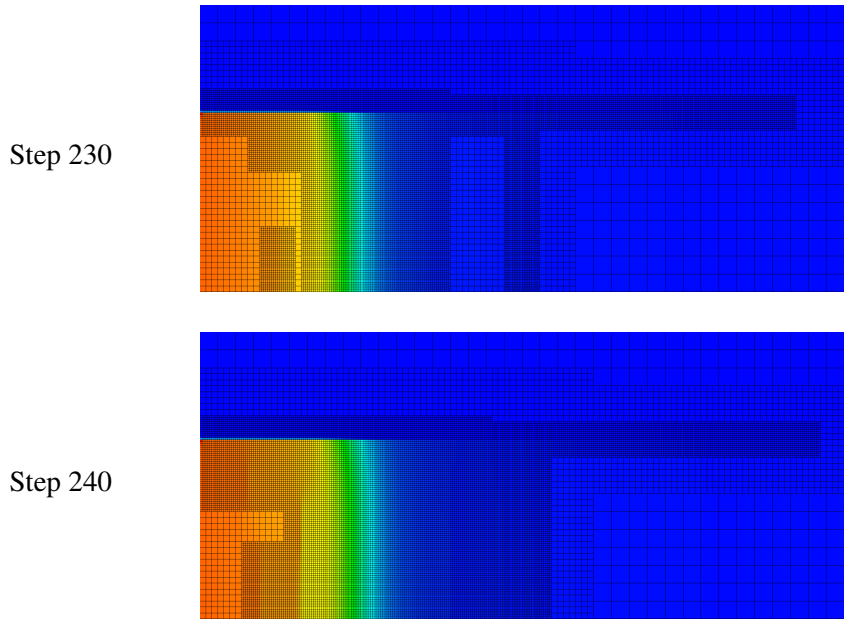

Figure 4: The initial flux limiter implementation generated serious mesh-imprinting on the solution which could in turn cause spurious refined patches. Two successive grid layouts and temperature fields from a thermal wave simulation are shown. Look for a nonphysical sharp front in the second plot at the location of a refinement boundary in the first plot.

ϕ (as well as mesh node locations, needed for metrics) into ghost layers around each patch. The non-AMR base code can then construct the limiter the same way it does for all other points.

Because the diffusion coefficient is modified based on the solution itself, these limiters introduce a nonlinear effect in an otherwise linear system. (The limiter is computed from a lagged solution so no true nonlinear solver is required.) We have found that it is important to take care in evaluating the limiter at coarse-fine interfaces, or solution artifacts may result. In particular, a solution front propagating across a coarse-fine interface may 'hang' temporarily at the interface, even as it continues to move elsewhere. The effect seems to start when the front is resolved at fine scale—which gives a steeper front and a larger gradient—but the resulting diffusion coefficient is used by a coarse cell. This kicks off a nonlinear feedback effect where the reduced diffusion coefficient makes the local front even steeper, until eventually enough of the front gets past the grid interface to start moving normally again.

The root cause traces back to the interpolation stencil used to fill ghost cells at the edge of a fine patch. Standard AMR techniques built into SAMRAI interpolate only from coarse data (including coarse cells covered by finer patches, which contain averaged fine data). Because other parts of the code require conservative interpolation and are sensitive to overshoots, the scheme is based on constructing limited slopes across each coarse cell. Near a front this approach can yield essentially piecewise-constant

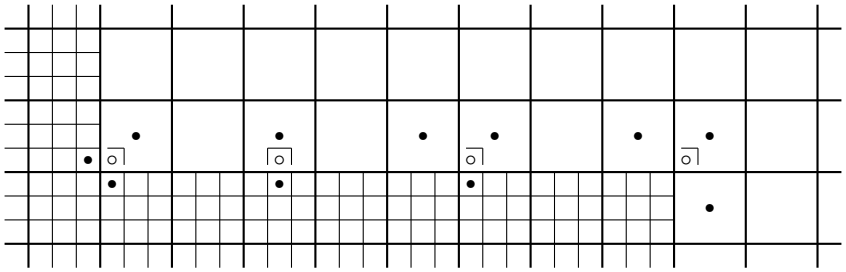

Figure 5: Flux limiter stencils in 2D: linear interpolation fills ghost zones (open circles) from surrounding data (closed circles). These ghost values will be used for constructing the scaled gradient at adjacent nodes on the coarse-fine interface, in the same way that real values are used for this purpose at interior nodes.

interpolation.

The priorities for constructing a diffusion flux limiter are different. The critical factor is not conservative interpolation but smoothness. (Overall conservation is still maintained—that's a separate issue from the limiter computation.) In order to obtain smoother limiter behavior we replaced the slope-based method with a scheme that interpolates *between* the coarse and fine data on either side of each ghost cell. The simplest case, where 2D reduces to 1D, is shown second from the left in Fig. 5.

It is a bit tricky to generalize this approach to all possible configurations of a coarse-fine interface because there are several constraints to observe. First, whenever a cell location is in the ghost layers of two different patches (where those layers intersect), each patch must compute the same interpolated value in that cell. Violating this condition would cause the different patches to compute different flux limiters, and thus different diffusion coefficients, for fluxes from one patch to the other, so diffusion would no longer be conservative. Second, the scheme must extend to all possible corner configurations in 2D and 3D. Third, we wanted the scheme to work with only a single layer of ghost data around each patch, since we did not want to increase communication costs. This constraint, combined with the first one above, eliminates a number of plausible uses of fine interior data that would not be available to all patches interpolating a particular ghost cell.

A rule that fits all constraints is to interpolate to a ghost cell using linear interpolation involving the coarse cell containing the ghost and all other cells (coarse or fine) with which the ghost cell shares a face. Fig. 5 shows the relevant stencils for 2D. Note that for the first, third, and fourth cases shown it is tempting to replace the 2D interpolation with a 1D interpolation using the fine cell to the lower left of the ghost cell. This would work for the grid arrangement shown in the figure, but even in 2D would require us to add two additional cases to account for the possibility that the "fine" cell in question might actually be coarse. In 3D the number of special cases multiplies beyond practicality. The rule using only face-neighbors is not necessarily the simplest for a particular ghost cell, but it appears to be simplest to implement because it holds special exceptions to a minimum.

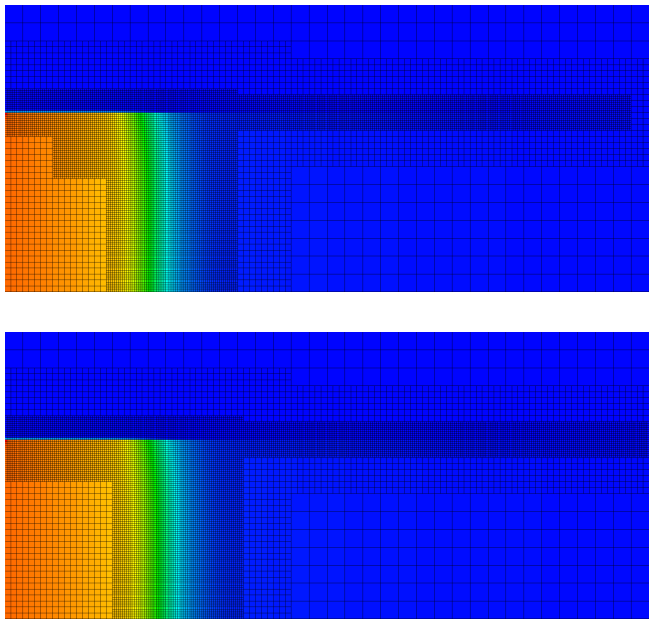

Figure 6: Improved interpolation stencils for the flux limiter yield a smoother thermal wave solution without spurious refinement.

We can in fact write down the relevant interpolation formulae in closed form (or nearly so) for any number of dimensions and with different refinement ratios in each dimension. The first step is to identify dimensions in which the ghost cell is centered in its containing coarse cell. An example is the second case in Fig. 5, where the ghost is centered in the horizontal dimension. These dimensions simply drop out of the formulae: there is no need for interpolation in a dimension where the ghost is centered.

Next, we interpolate an intermediate value from the containing coarse cell ϕ_0^c and all of the other coarse cells that share a face with the ghost cell. The coefficient for the coarse neighbor ϕ_i^c in dimension i is $(r_i - 1)/2r_i$, where r_i is the refinement ratio in dimension i. The coefficient for the containing coarse cell is then $1 - \sum(r_i - 1)/2r_i$. So the value we want can be written as

$$\phi_0^f = \phi_0^c + \sum_{i \in N_c} \frac{r_i - 1}{2r_i}(\phi_i^c - \phi_0^c), \tag{3}$$

where i is limited to dimensions where the ghost cell has coarse neighbors. In Fig.5, this step solves the fourth example completely (all neighbors are coarse, so ϕ_0^f is the interpolant we want), and it reduces the third case to the equivalent of the second by eliminating the horizontal dimension. If there are no coarse neighbors, as in the first and second examples, then $\phi_0^f = \phi_0^c$ and we immediately go to the next step.

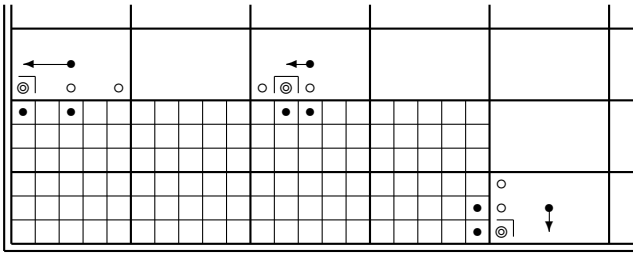

Figure 7: Some special cases in flux limiter interpolation arise for refinement ratios greater than three and at physical boundaries.

We now turn to the dimensions where the ghost cell has fine neighbors ϕ_i^f. The desired interpolant is

$$\phi^f = \frac{2\phi_0^f + \sum_{i \in N_f}(r_i - 1)\phi_i^f}{2 + \sum_{i \in N_f}(r_i - 1)}. \tag{4}$$

I have skipped over some special cases: If the refinement ratio r_i in a given dimension is greater than three, then there can be ghost cells that are neither centered in dimension i, nor adjacent to a coarse cell face in that dimension. There doesn't seem to be as elegant a solution for this "intermediate" case. What I do is to extrapolate from ϕ_0^c using slope information in dimension i, eliminating this dimension and obtaining an effective ϕ_0^c to go into Eq. 3. For purposes of Eqs. 3 and 4 the "intermediate" case then looks much like the "centered" case: dimension i is contained in neither N_c or N_f. In 2D the fine values in the adjacent fine patch are available, unique, and suitable for constructing the needed slope. In 3D the adjacent fine values may not be unique, and so instead I make a first pass over the ghost cells where this intermediate issue doesn't apply, that can be handled directly using the algorithm above. Then I use these ghosts to construct the necessary slopes.

Another special case comes up when a ghost cell is up against a face of its containing coarse cell in dimension i, but the other side of that face is a physical boundary. Rather than tie into the code defining the physical boundary conditions, I treat this more like the intermediate case above. Using slope information for dimension i I extrapolate an effective ϕ_0^c towards the boundary and then proceed with Eqs. 3 and 4 for the other dimensions.

Fig. 7 illustrates these special cases. In each example shown we need to interpolate to the ghost cell marked with a double circle. The middle case is neither centered nor has a neighbor cell in the horizontal direction. So we adjust the coarse value of the containing cell using horizontal slope information to obtain a modified ϕ_0^c, as suggested by the arrow. In 2D the two fine cells below the target provide a unique slope, but in 3D the fine adjacent cells may not be unique and we instead obtain a slope from the interpolated ghost cells shown with open circles. In either case we have reduced the problem to one like the second example in Fig. 5, and we then proceed with Eqs. 3 and 4. (The reason we do not simply interpolate between the two open circles is that we still want to maintain the close coupling to the fine cell directly

below the target.) The other two examples in Fig. 7 work the same way except that in these cases the target ghost cell is adjacent to a physical boundary.

Similar circumstances arise near reduced or enhanced connectivity points. I won't go through these complications in full detail, but the key principle is to always obey the conservation requirement that any two patches computing the flux limiter—and thus the diffusion coefficient—at the "same" location must do it in exactly the same way using exactly the same data. Equivalent diffusion coefficients must always be equal. So, for example, if one sector touching a reduced connectivity point is fine and the other two are coarse, then it is appropriate to interpolate into each of the two orthogonal ghost cells bordering the fine patch at the corner as if the "missing" quadrant were outside the physical boundary. If two of the sectors are fine and the other is coarse, though, then we interpolate into the orthogonal ghost cells of both fine patches (which represent the same location) using the formula for a ghost with two fine neighbors, as in the example on the left end of Fig. 5. This way both fine patches fill the ghost cell with the same value. The actual gradient computation in these cases is handed off to native non-AMR code that understands reduced connectivity points, so the diagonal ghost cells will be ignored and do not need to be interpolated.

Optimization and Scaling

The data structures for managing coarse-fine interface stencils are complex. The initial implementation built them from scratch for every linear system solve, which proved to be a significant added expense. To avoid this, the structures are now built only once per AMR regrid cycle, and are thus re-used for multiple physics packages and timesteps. (The AMR mesh is reconstructed at user-specified intervals, typically once every 10 timesteps.) It is necessary to reload the structures with new coefficients for each solve but this takes roughly a tenth the time of building them in the first place. The overhead cost of dealing with these structures is then well below that of the linear solver itself.

The solver interface was designed to take full advantage of MPI parallelism as used in SAMRAI and the native code. To demonstrate weak scaling we use a tiling approach where similar grid configurations are replicated a variable number of times. Figure 8 shows a 4-tile version of the mesh used for 2D scaling runs. 3D tests used tiles with similar numbers of grids as the 2D cases, but in 3D the numbers of cells per grid and of surface faces at the coarse-fine interface were both significantly larger. These differences may have contributed to minor differences in the scaling results obtained for 2D and 3D.

The timings shown in Figs. 9 and 10 were obtained on *zin,* a Xeon-based Linux cluster with 16 processors per node and InfiniBand QDR high speed interconnect, using the *hypre* alpha release 2.8.3a. Each test ran for 10 timesteps without regridding. The timings do not include initialization of the physics code itself, but do include the costs of initializing the diffusion data structures once for the given AMR mesh layout and then refilling them with new coefficients for the other nine timesteps. These are preliminary results, but have already led to improvements in *hypre*. Further improvements are likely as we explore the effects of various *hypre* algorithmic options and adjust AMR parameters to tune grid size and data locality for better solver performance.

Overheads for both the 2D and 3D tests are under control. The most expensive phase in all cases ("solveSystem") consists of *hypre* solving the linear system using AMG (algebraic multgrid) as a

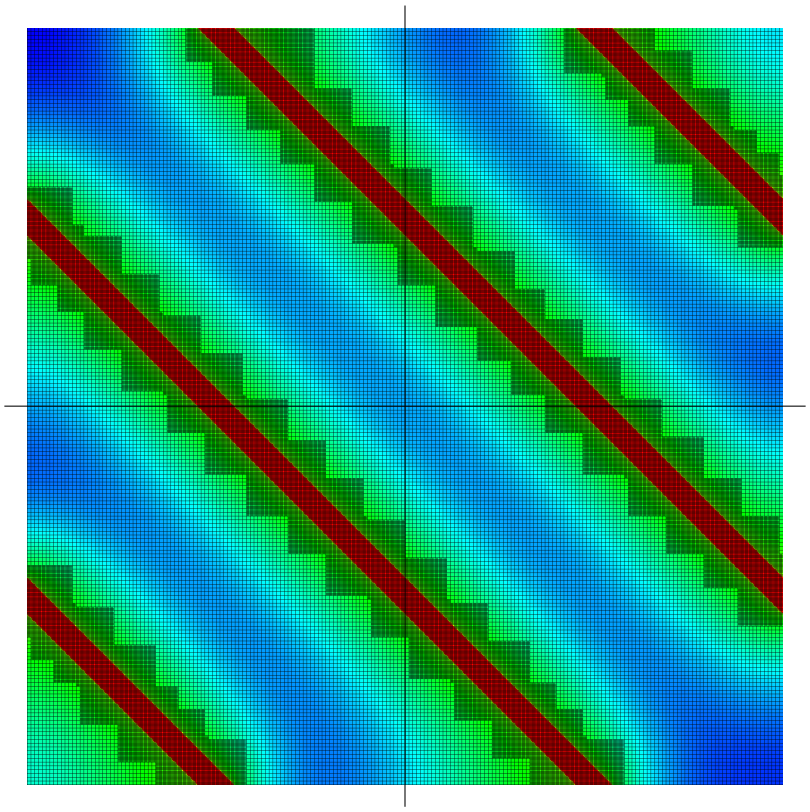

Figure 8: The weak scaling tests are tiled so that size increases with processor count. This 2D example has 4 tiles (2×2) and would be run on 4 processors.

preconditioner to GMRES. The second most expensive phase is where *hypre* generates its internal data structures to support multigrid—this phase is matrix-dependent and so must be repeated for every solver call because the diffusion coefficients keep changing. Other *hypre*-related expenses shown in the figures are for building the *hypre* grid, graph, matrix, and vector objects ("buildHypreStructs")—these structures must be rebuilt when the mesh layout changes but can be re-used for different coefficients. The lines marked "setValues" show the expense of loading new values into the *hypre* matrix and vector objects from the data structures discussed in this paper. The expense of building those structures and loading them with coefficients is shown as "interfaceStructures". Finally, the lines marked "radiationPrep" show the work done mostly in native non-AMR code to compute metrics, diffusion coefficients, the flux limiter, and so on. In both 2D and 3D the overhead expenses seem to be scaling as well as or better than the solver and are considerably cheaper, both of which are good signs and suggest that efficiency of the solver itself will be the main focus for future performance improvements.

In 2D the solver times begin creeping up for more than 2000 processors. 3D results, on the other hand, seem quite flat. The difference may in part be due to the relatively small grids in the 2D tests, compared to the larger ones in 3D that require more work per grid. It will be interesting to try other test problems and

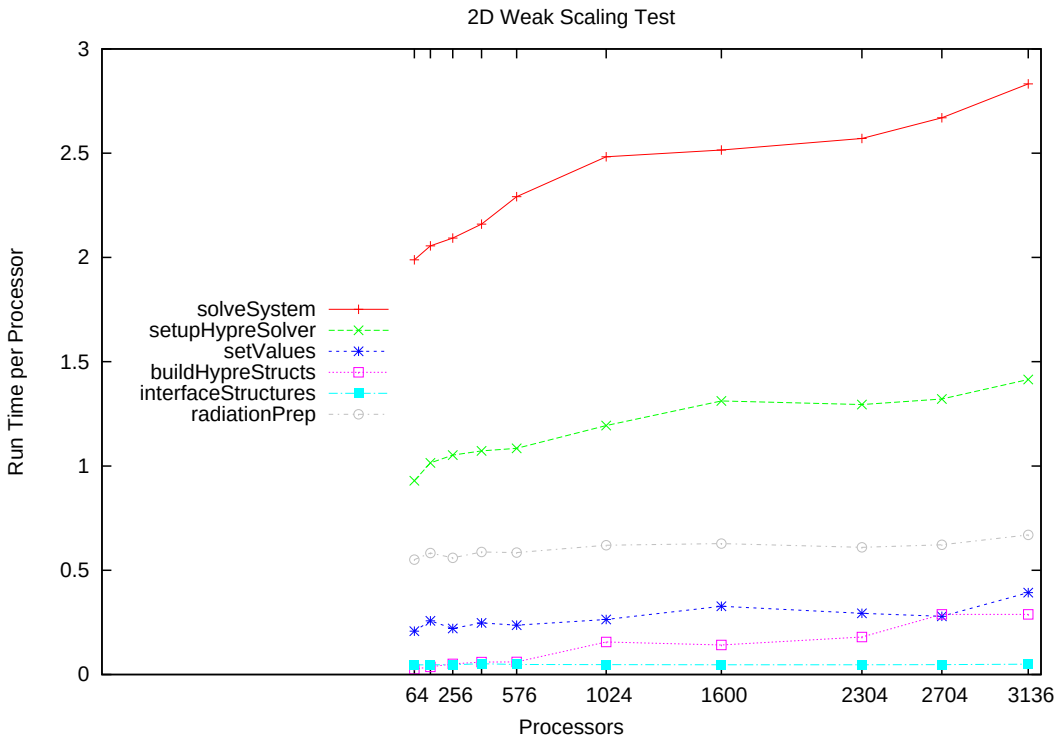

Figure 9: Results for 2D weak scaling show good but not perfect scalability.

experiment with more processors and different machine configurations.

Experience with other codes suggests that issues of locality and data layout will be crucial for obtaining better performance. So far we have been using mesh parameters chosen based on experience with other parts of the code, particularly explicit hydrodynamics. *hypre* inherits both the mesh and the decomposition among processors from the main code—it does not redistribute its own data internally, but relies on the calling program to use an efficient partition strategy. Given that solving several implicit linear systems at each timestep is a major expense, it will make sense to explore AMR parameters and options with an eye towards optimizing solver performance.

Acknowledgments

Help and advice were received from many members of the SAMRAI and application code teams. Some of the decks used for simulations were based on earlier versions by John Hayes and Robert Anderson.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-609323.

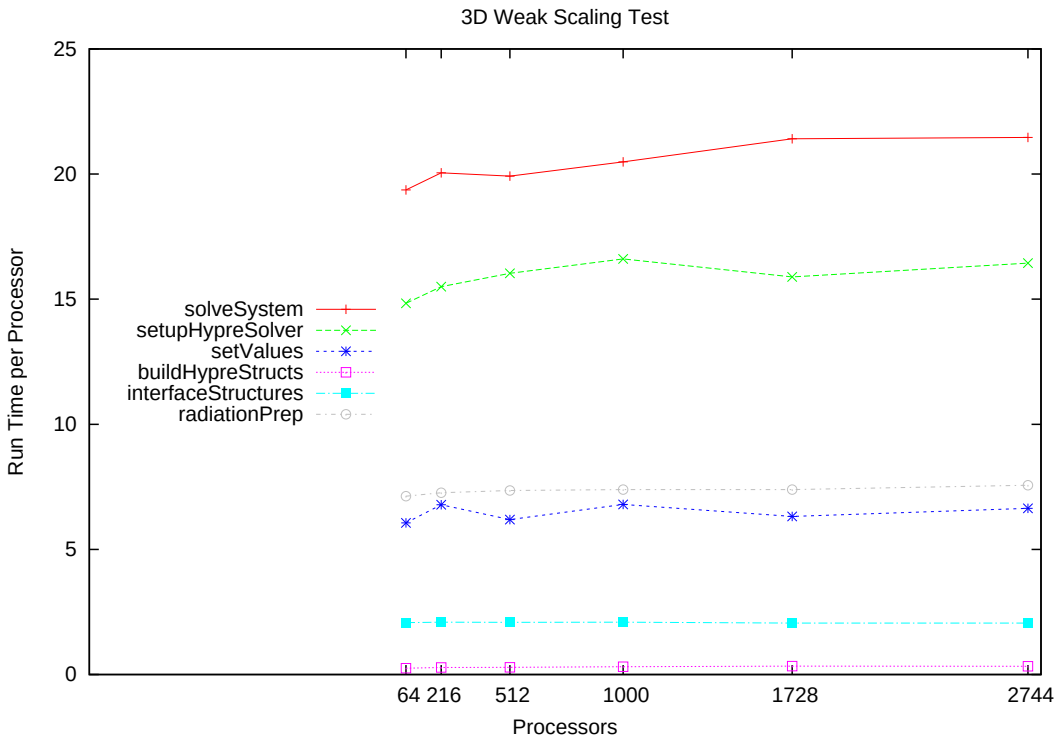

Figure 10: 3D weak scaling results seem better than those for 2D, perhaps because the 3D stencils and the test problem used require more work per grid.

References

[1] Almgren, A. S., J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome A Conservative Adaptive Projection Method for the Variable Density Incompressible Navier-Stokes Equations. *Journal of Computational Physics* **142**:1–46 (1998).

[2] Bell, J., M. Berger, J. Saltzman, and M. Welcome Three-dimensional Adaptive Mesh Refinement for Hyperbolic Conservation Laws. *SIAM Journal of Scientific Computing* **15**:127–138 (1994).

[3] Gentile, N. A. Implicit Monte Carlo Diffusion—An Acceleration Method for Monte Carlo Time-Dependent Radiative Transfer Simulations. *Journal of Computational Physics* **172**:543–571 (2001).

[4] Graziani, F. and J. LeBlanc The Crooked Pipe Problem. Lawrence Livermore National Laboratory, Livermore, CA, UCRL-MI-143393 (2000).

[5] Howell, L. H. and J. A. Greenough Radiation Diffusion for Multi-fluid Eulerian Hydrodynamics with Adaptive Mesh Refinement. *Journal of Computational Physics* **184**:53–78 (1981).

[6] Pert, G. J. Physical Constraints in Numerical Calculations of Diffusion. *Journal of Computational Physics* **42**:20–52 (1981).

[7] Wickett, M. E. Mesh Relaxation, Mesh Motion Controls, and Adaption Criteria with ALE-AMR in an ASC Code. In *Proceedings of the 16th Nuclear Explosives Code Developers Conference*, Los Alamos, NM, October 18–22, 2010 (2011).

[8] Zhang, W., L. Howell, A. Almgren, A. Burrows, and J. Bell CASTRO: A New Compressible Astrophysical Solver, II. Gray Radiation Hydrodynamics. *Astrophysical Journal Supplement Series* **196**:20 (2011).

[9] `https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html`

[10] `https://computation.llnl.gov/casc/SAMRAI/`